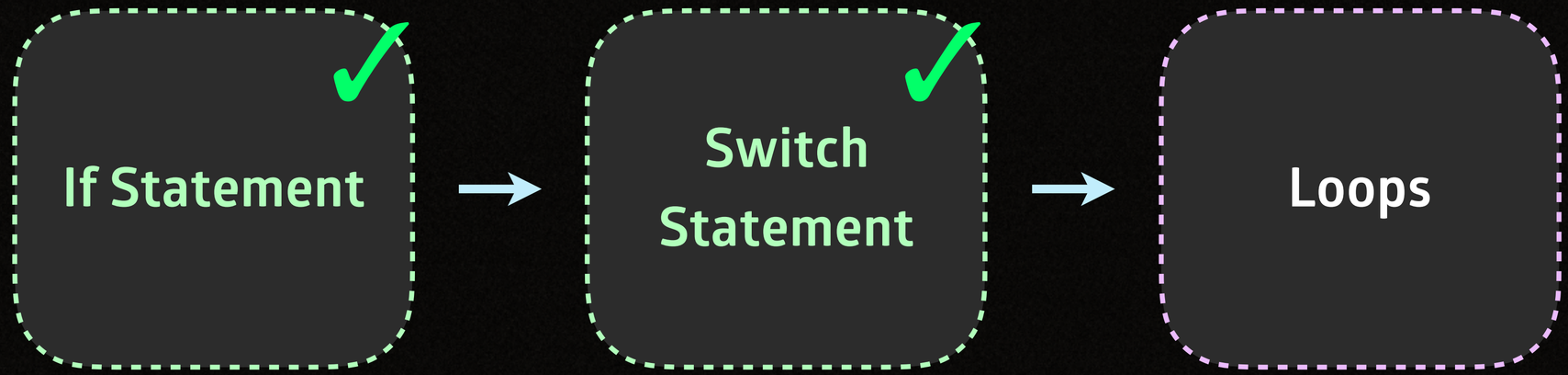
PART III

Fundamentals

Control Flow



Why?

Why?

Humans are **very bad**
at doing **the same set of tasks**
over and over again...

Why?

Computers are **very good**
at doing **the same set of tasks**
over and over again!

Do this millions of times, OK? OK?!

Shut up!...
And gimme a loop!

LOOPS

Time to learn how to control the execution flow using loops

For Statement → 🍀 Lucky Number → 🕵️ Word Finder → Labeled Statements

FOR STATEMENT

Time to learn how to control the execution flow using loops

For Statement → Nested Loops → Range Loops

FOR STATEMENT

Repeats a block of code as long as its condition is true

```
for i := 1; i <= 5; i++ {
    sum += i
}
```

INIT STATEMENT

Runs once. Before the for loop begins.

init statement

```
for i := 1; i <= 5; i++ {
    sum += i
}
```

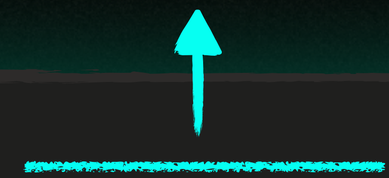
simple statement → declares i → runs once

CONDITION EXPRESSION

It will be checked just before each loop step starts

condition

bool expression



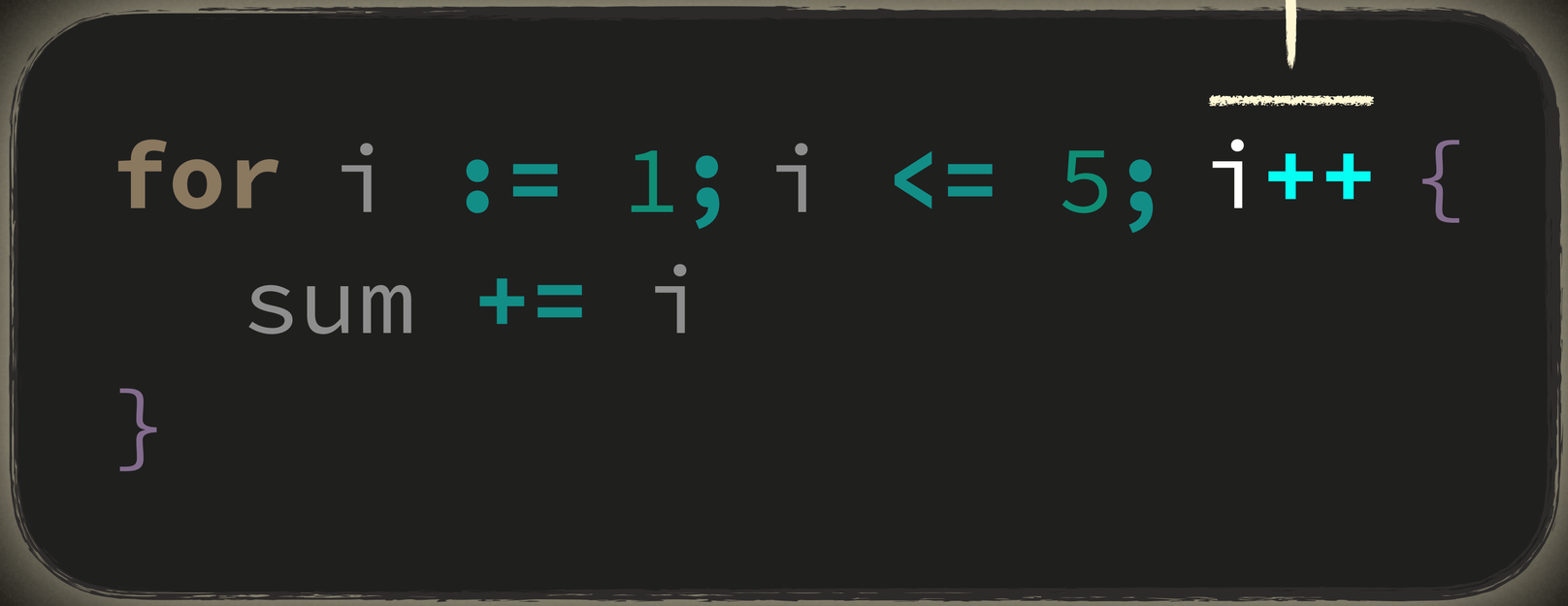
```
for i := 1; i <= 5; i++ {
    sum += i
}
```

*The loop will **continue** as long as this expression is **true***

POST STATEMENT

***Runs after** each step of the loop*

post statement
simple statement



```
for i := 1; i <= 5; i++ {
    sum += i
}
```

SEPARATORS

They separate the parts of a for statement

```
for i := 1; i <= 5; i++ {
    sum += i
}
```

FOR'S BLOCK

Statements inside the block runs only if the condition yields true

```
for i := 1; i <= 5; i++ {
    sum += i
}
```

*runs the **block** as long as the **condition** is **true***

loops.....until i > 5

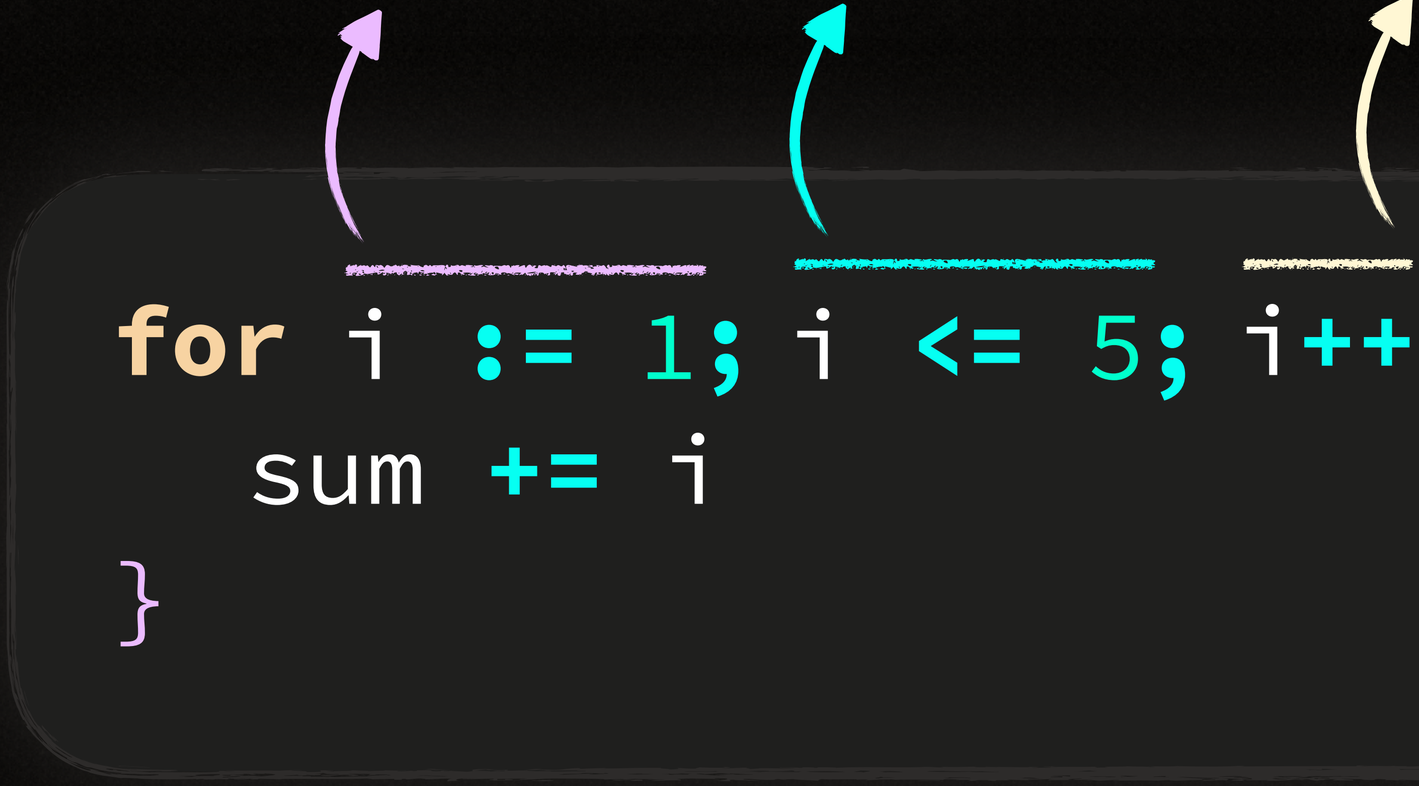
SUMMARY

All of those are optional

init statement
simple statement

condition
bool expression

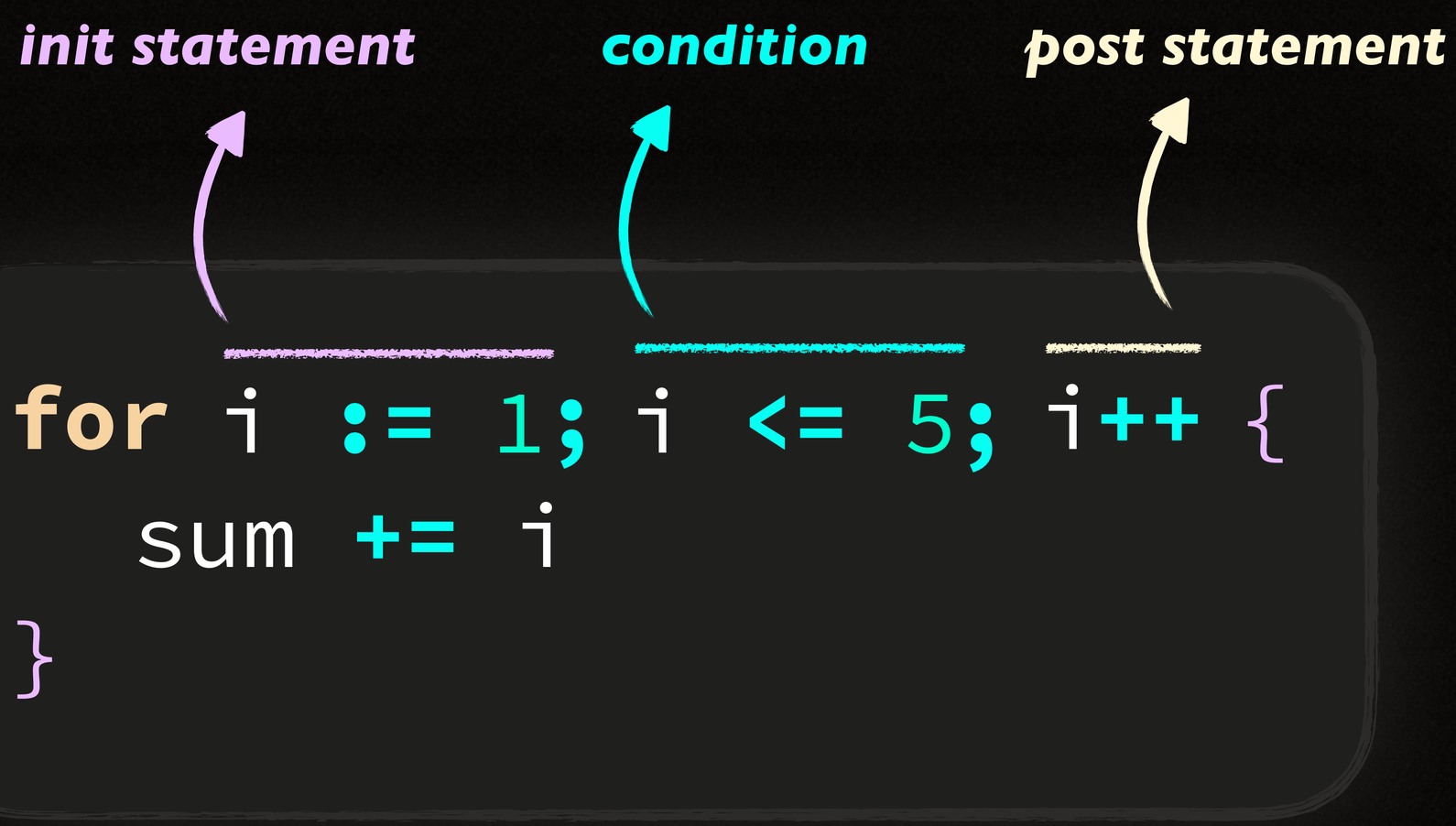
post statement
simple statement

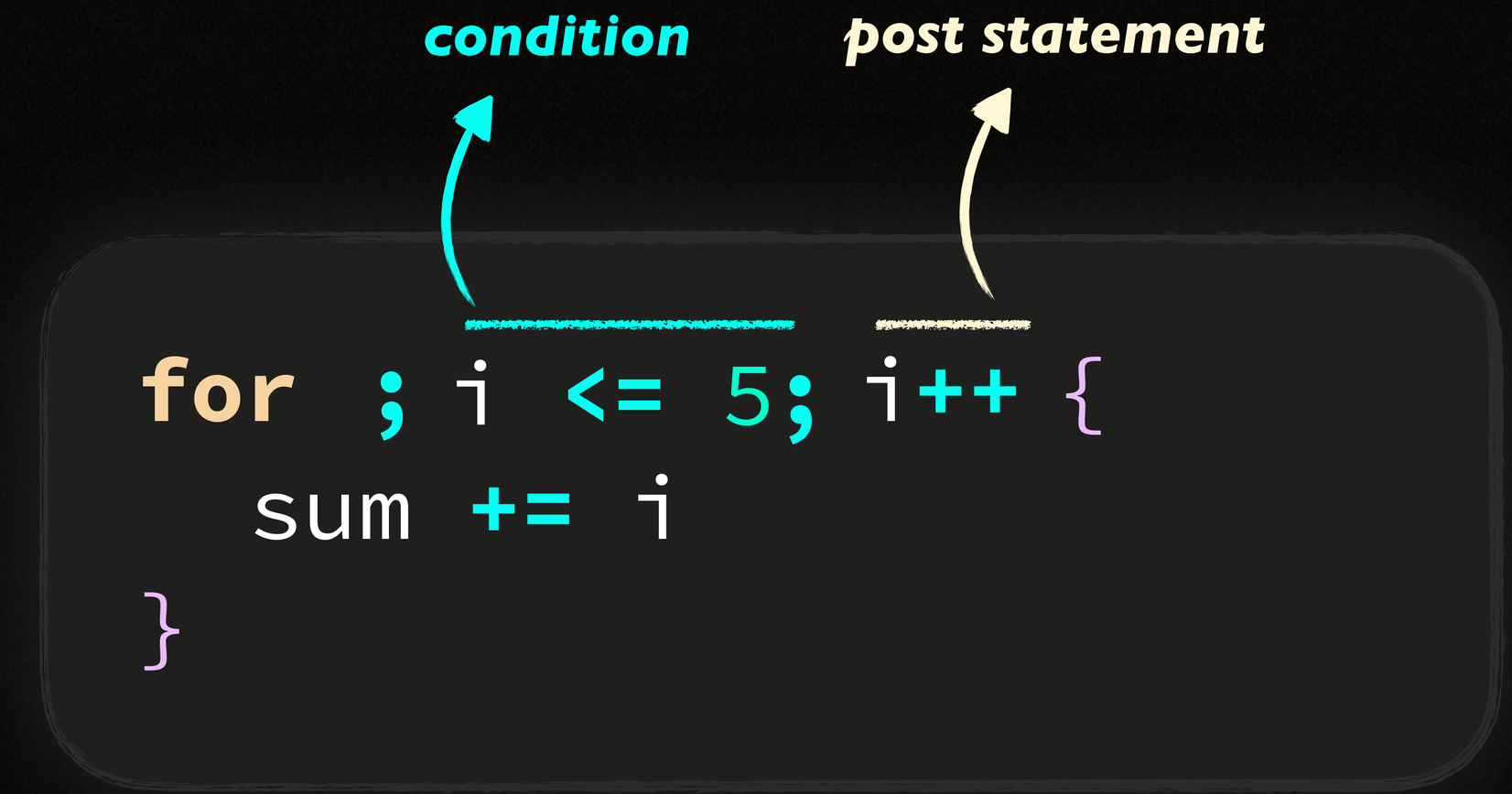


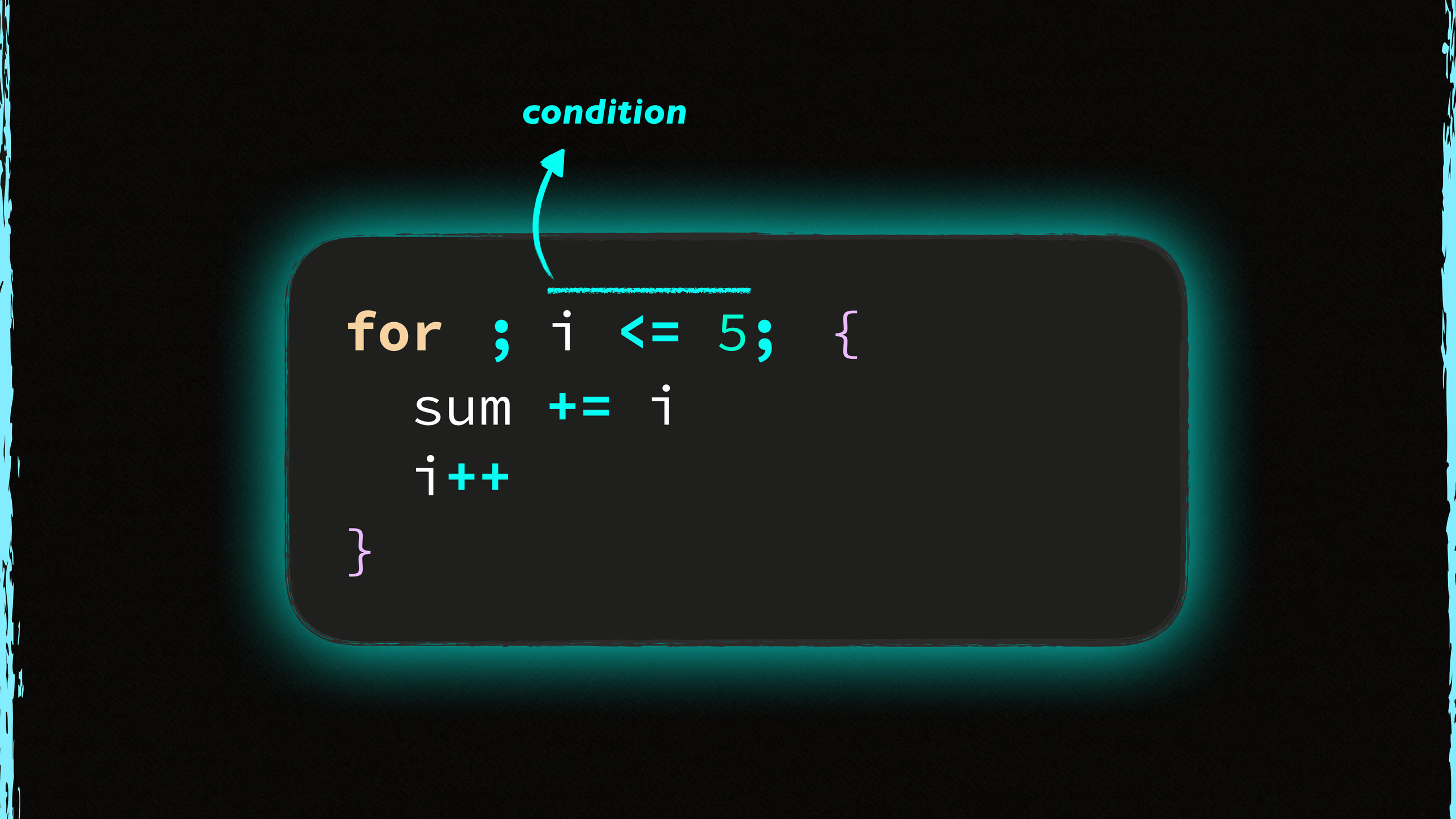
```
for i := 1; i <= 5; i++ {
    sum += i
}
```

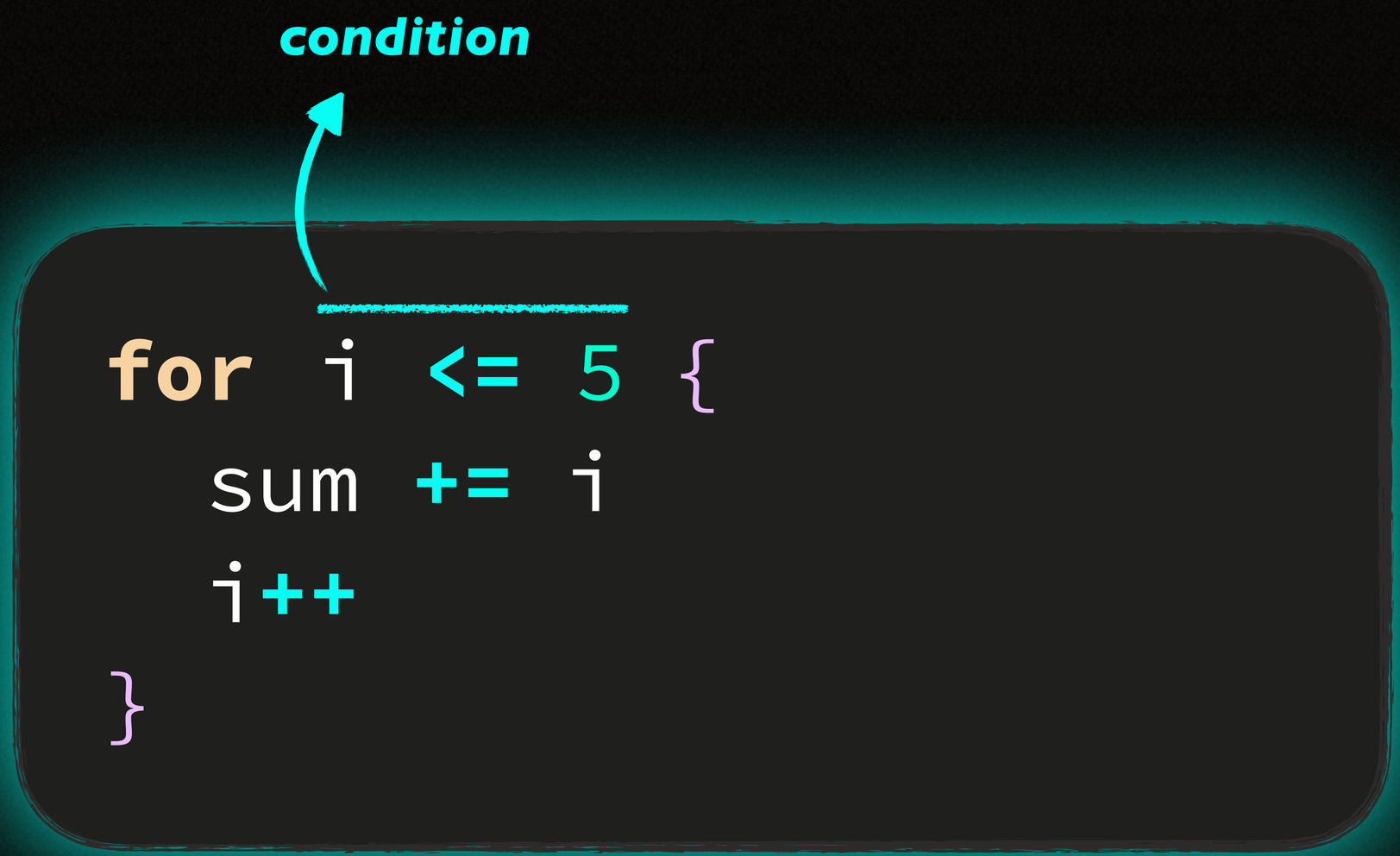
BREAK STATEMENT

Ends the loop immediately


init statement
condition
post statement
for i := 1; i <= 5; i++ {
sum += i
}

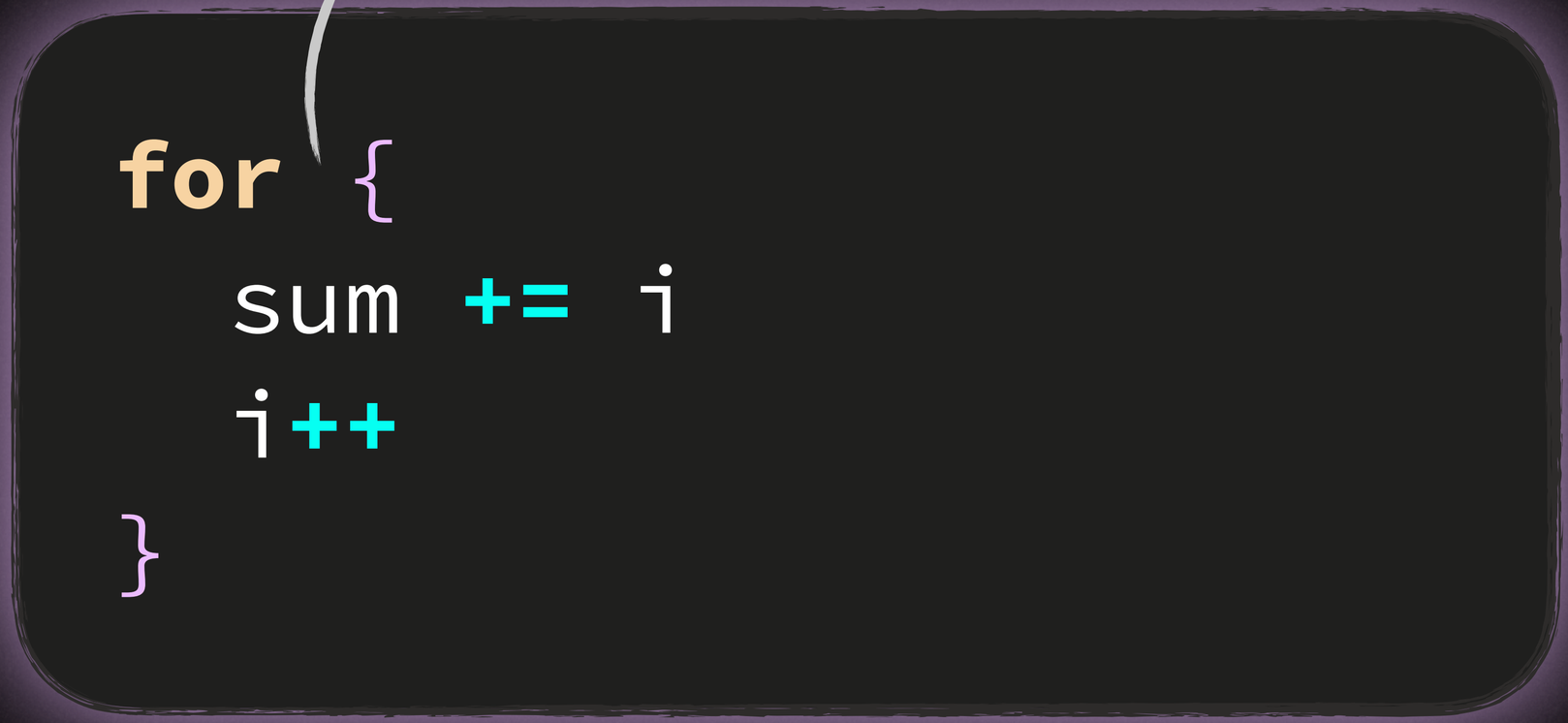

condition
post statement
for ; i <= 5; i++ {
sum += i
}


condition
for ; i <= 5; {
sum += i
i++
}



this is a for statement with a single-condition

condition *is still here, just hidden, it equals to* ***true***



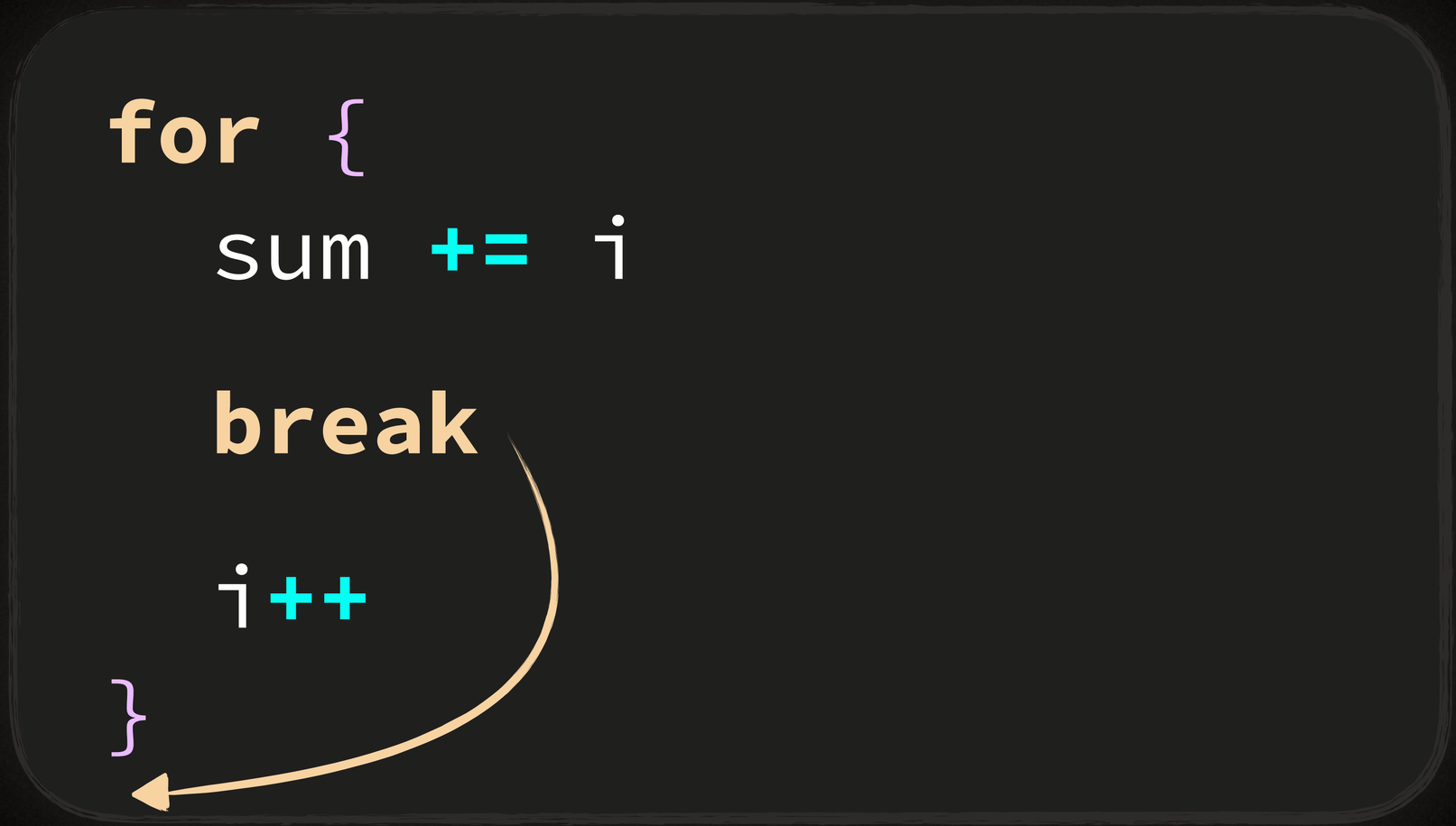
```
for {
    sum += i
    i++
}
```

this is an infinite loop

∞

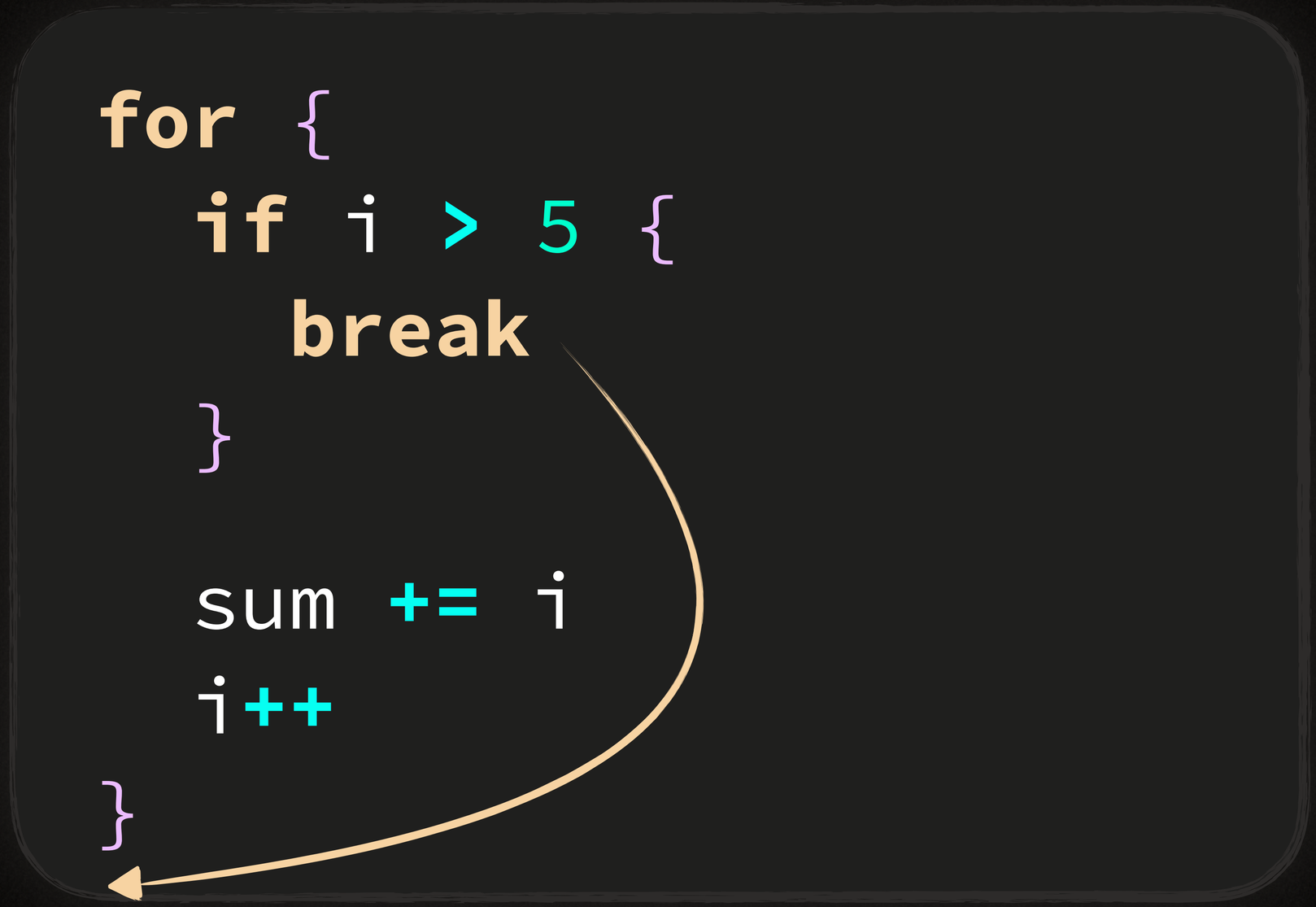
BREAK STATEMENT

You can exit from the loop by using the break statement



BREAK STATEMENT

You can exit from the loop by using the break statement



CONTINUE STATEMENT

Quits the current step and repeats the loop from where it left off

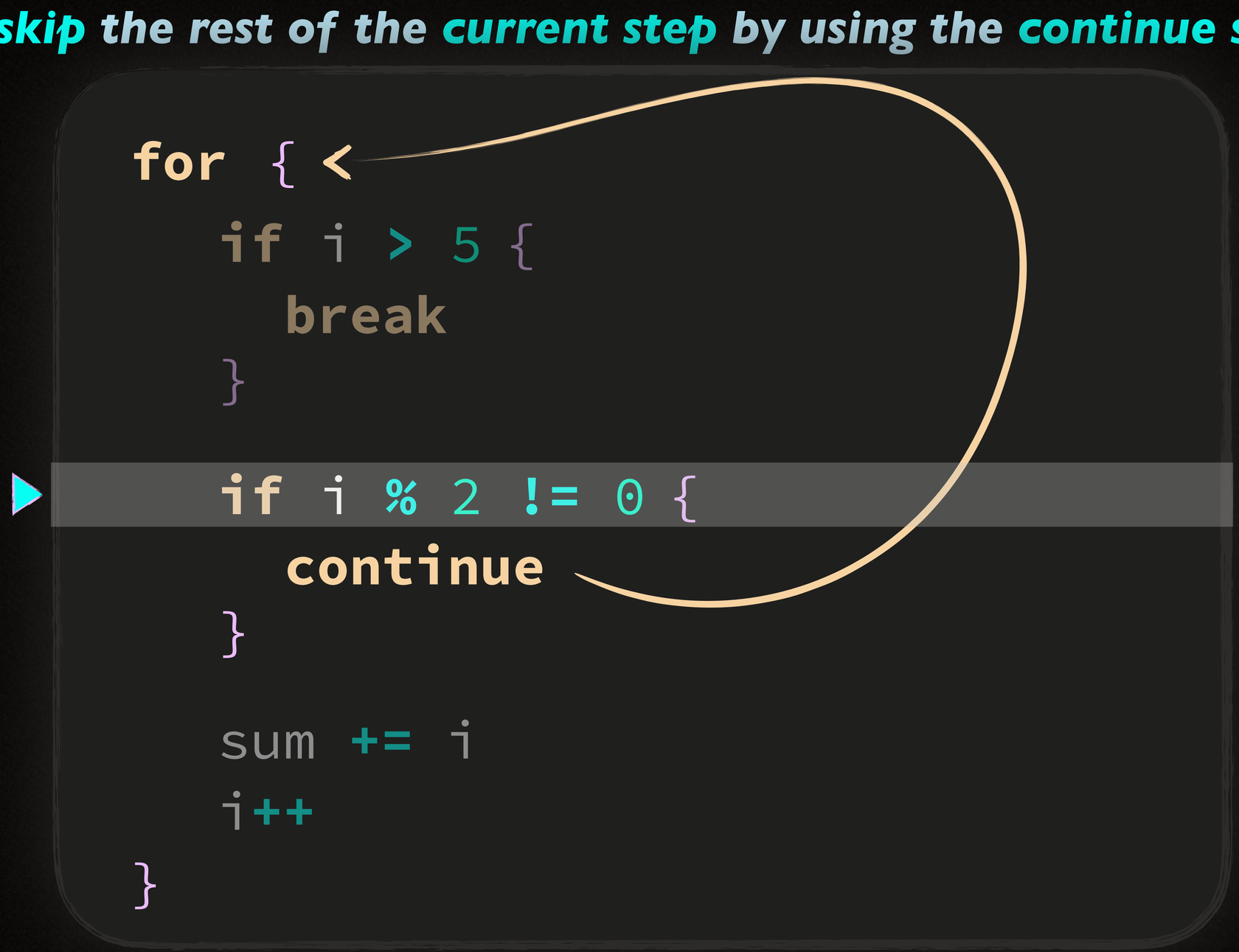
CONTINUE STATEMENT

Previously....

```
for {
    if i > 5 {
        break
    }

    sum += i
    i++
}
```

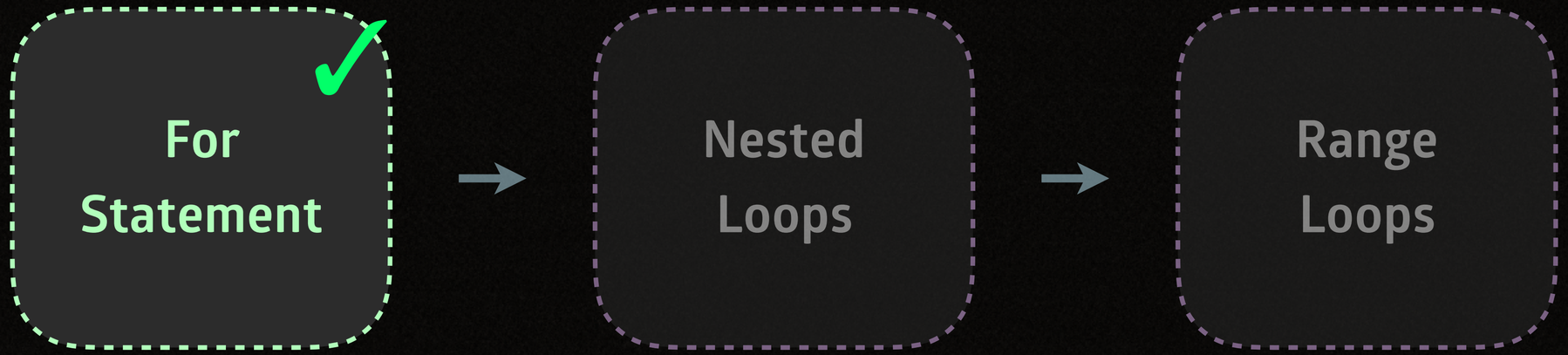
CONTINUE STATEMENT

You can skip the rest of the current step by using the continue statement



FOR STATEMENT

Congrats! You've learned how to use loops in Go!



NESTED LOOPS

Let's put a loop inside another loop

X	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	1	2	3	4	5
2	0	2	4	6	8	10
3	0	3	6	9	12	15
4	0	4	8	12	16	20
5	0	5	10	15	20	25

Header of the table

X	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	1	2	3	4	5
2	0	2	4	6	8	10
3	0	3	6	9	12	15
4	0	4	8	12	16	20
5	0	5	10	15	20	25

Vertical header of the table

X	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	1	2	3	4	5
2	0	2	4	6	8	10
3	0	3	6	9	12	15
4	0	4	8	12	16	20
5	0	5	10	15	20	25

Parent loop

Nested loop

X	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	1	2	3	4	5
2	0	2	4	6	8	10
3	0	3	6	9	12	15
4	0	4	8	12	16	20
5	0	5	10	15	20	25

✓

X	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	1	2	3	4	5
2	0	2	4	6	8	10
3	0	3	6	9	12	15
4	0	4	8	12	16	20
5	0	5			20	25

CHALLENGE!

CREATE A DYNAMIC TABLE

- Get the **size of the table** from the command-line

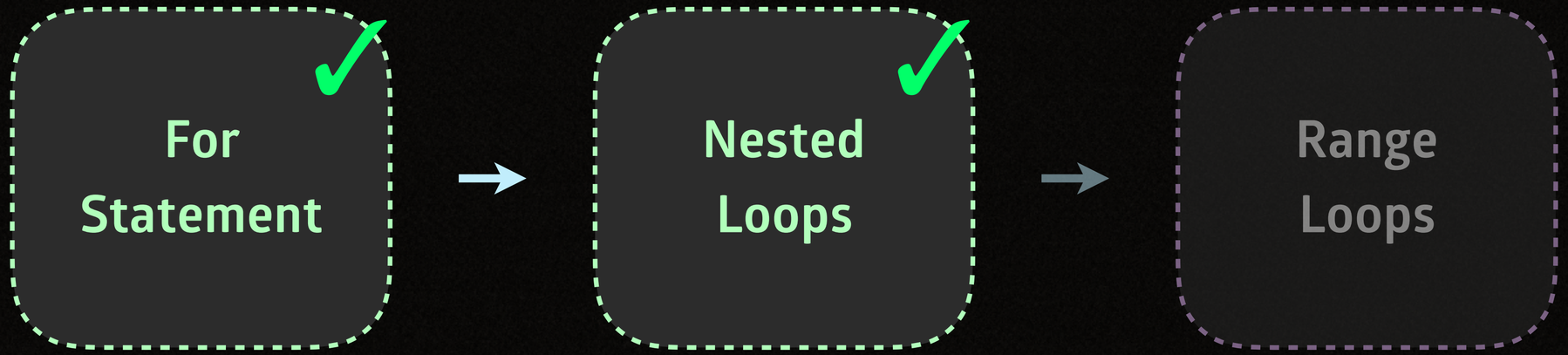
- Passing **5** should create a **5x5** table
- Passing **10** for a **10x10** table

BONUS

- Create division, addition and subtraction tables

FOR STATEMENT

Congrats! You've learned how to use nested loops!

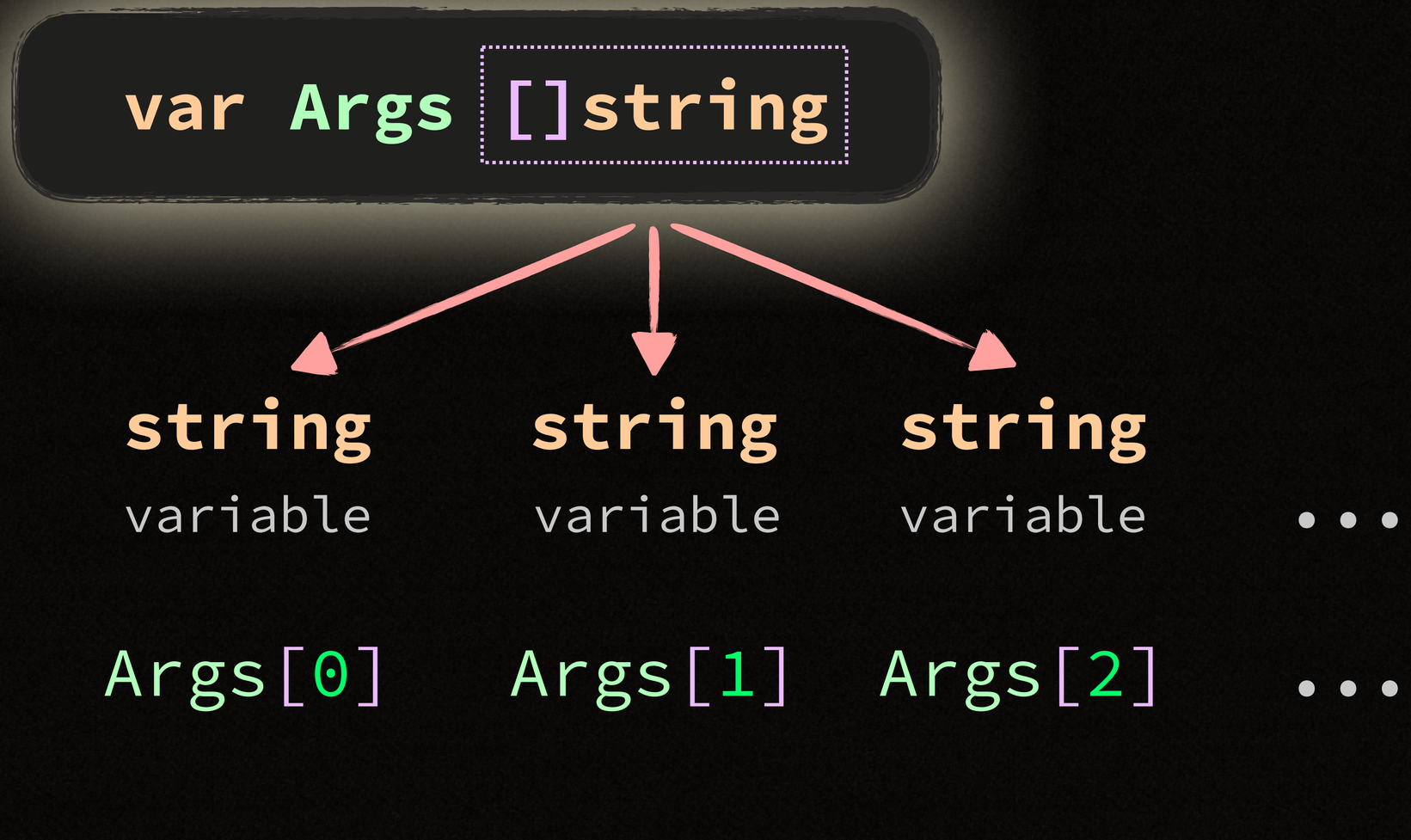


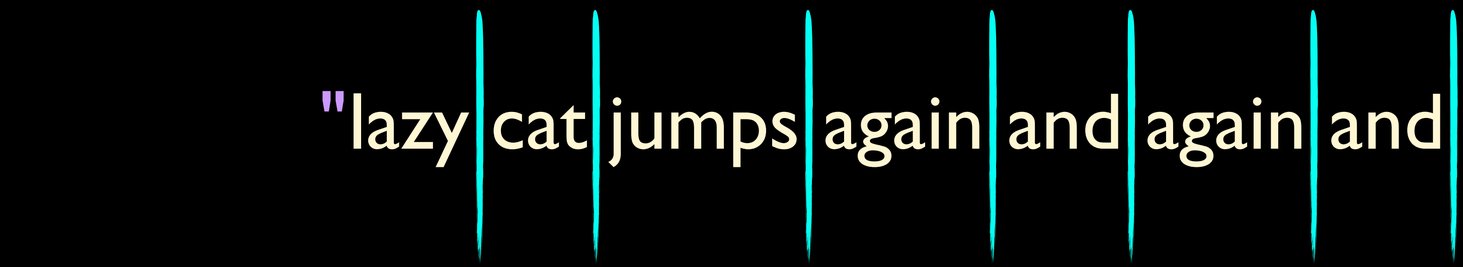
LOOPING OVER SLICES

Let's simply loop over a slice

os.Args

a slice stores unnamed variables inside





strings.Fields() **splits** this string from the white-space points

[]string

{ "lazy", "cat", "jumps", "again", "and", "again", "and", "again" }

each one becomes an unnamed string variable

FOR RANGE

Easy way to loop over slices

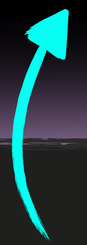
```go
for i, v := range words {
    fmt.Println(i, v)
}
```

range clause

```
for i, v := range words {
    fmt.Println(i, v)
}
```

```go
for i, v := range words {
    fmt.Println(i, v)
}
```

index variable

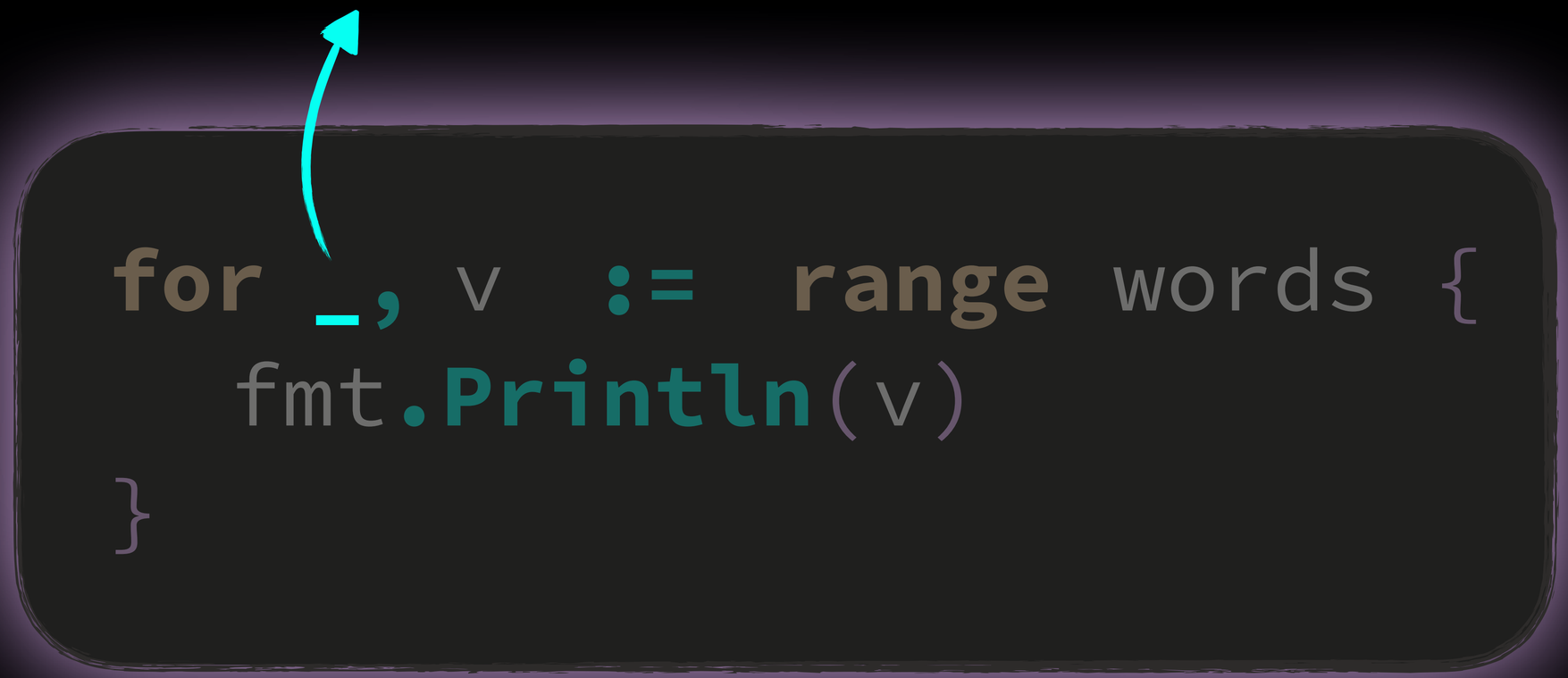


```
for i, v := range words {
    fmt.Println(i, v)
}
```

*This variable is **optional** and it **starts** from 0 to **len**(words) - 1*

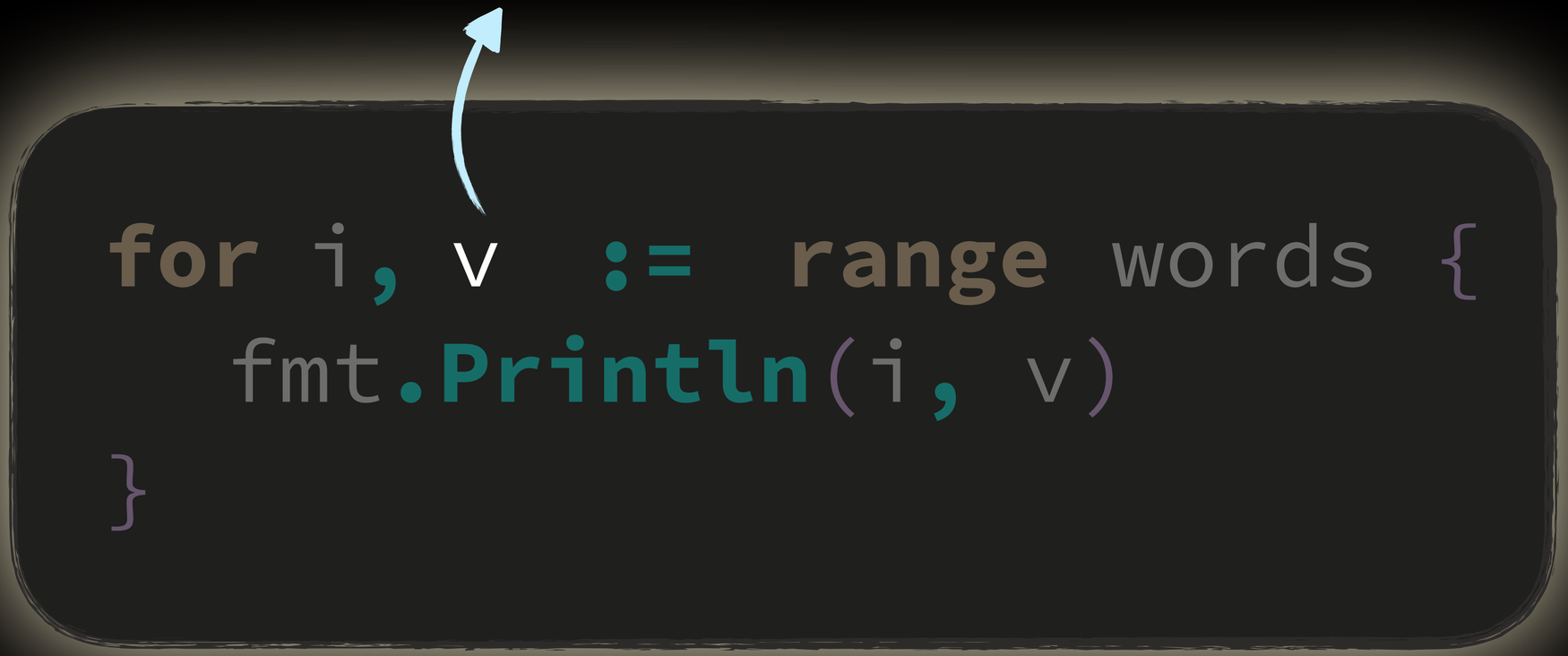
*if i < **len**(words)*

Blank-identifier *can discard the* ***index variable***



```
for _, v := range words {
    fmt.Println(v)
}
```

value variable

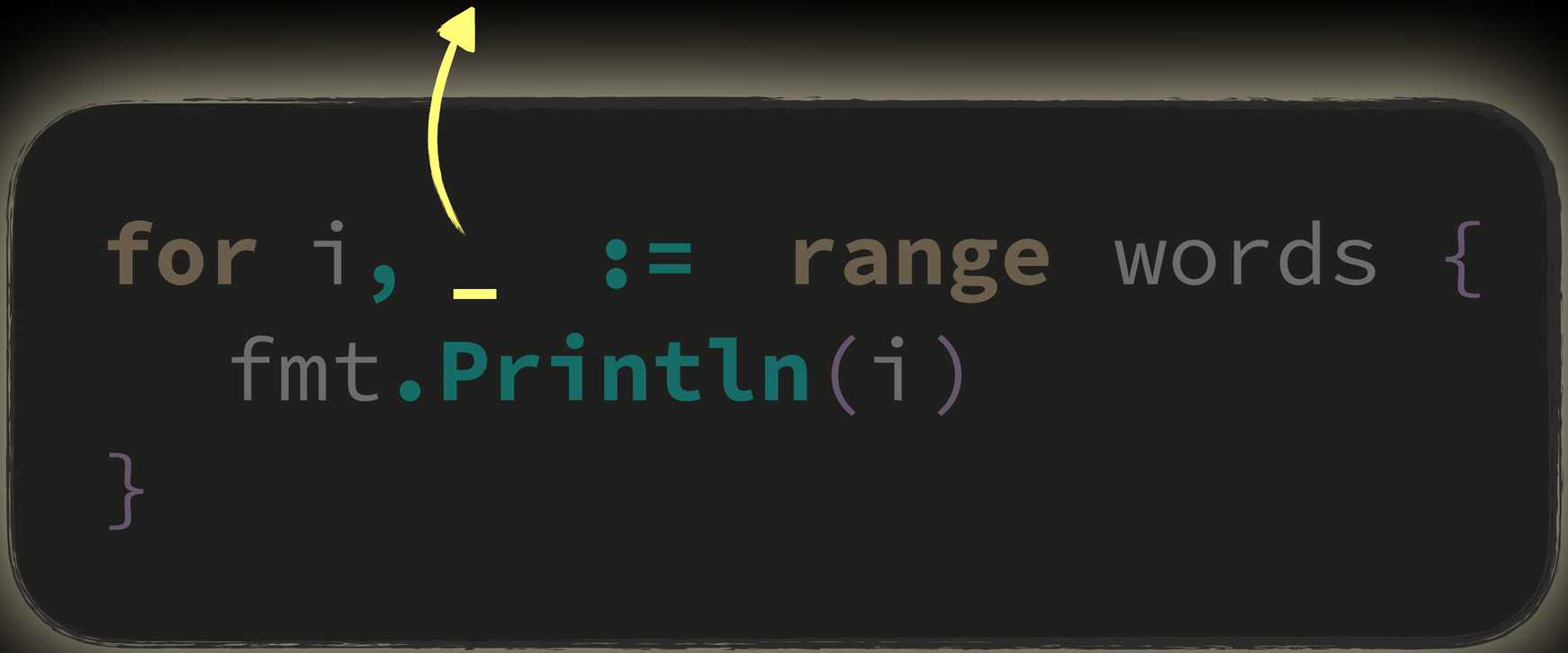


```
for i, v := range words {
    fmt.Println(i, v)
}
```

*This variable is also **optional** and on each step **range assigns** the **next value** to **v***

v = words[i]

Blank-identifier can also discard the values



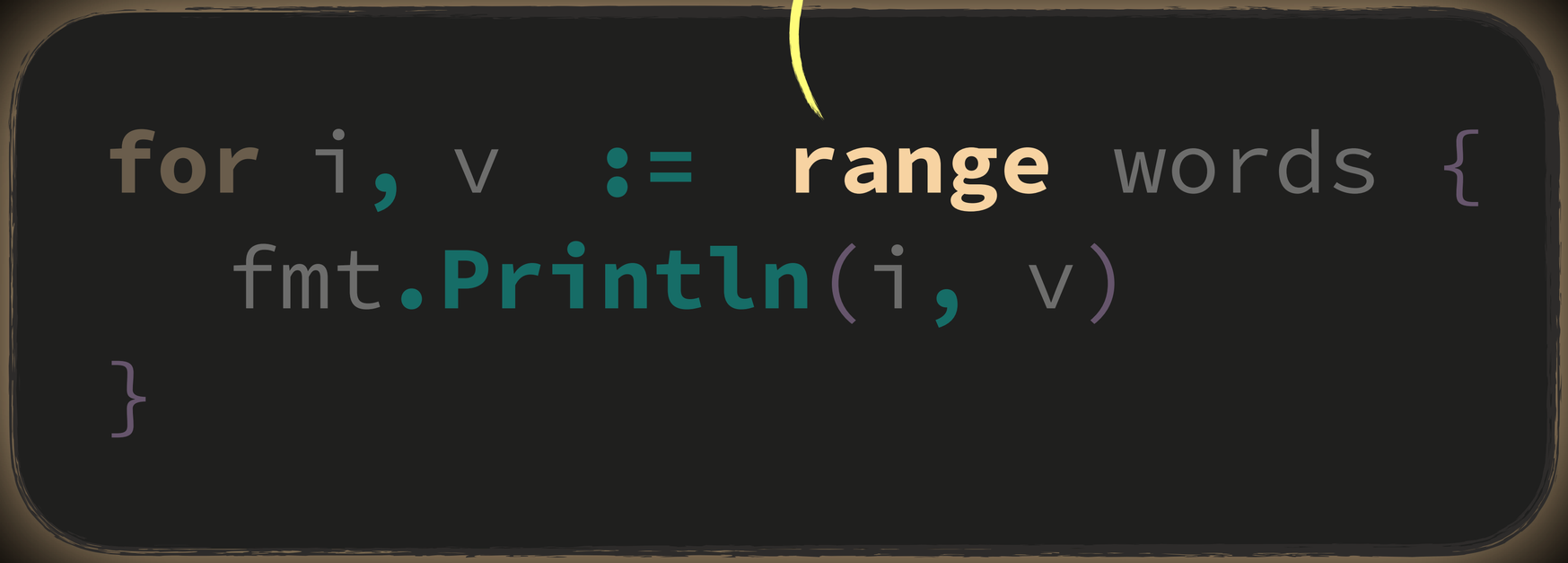
```
for i, _ := range words {
    fmt.Println(i)
}
```

```
for  i  :=  range words {
    fmt.Println(i)
}
```

*If what you need is **only the index values***

*Then you can **remove the value variable** entirely*

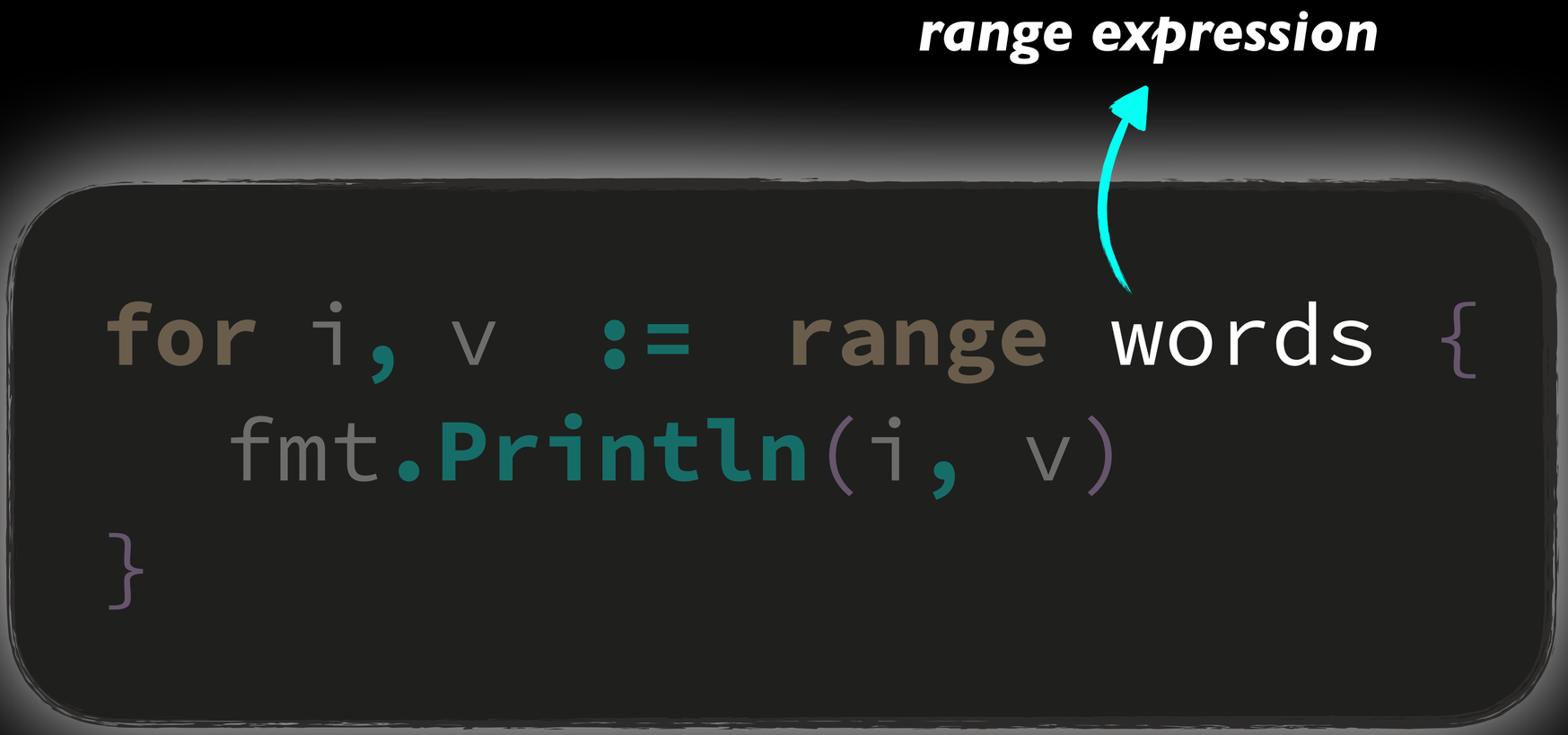
range keyword



```
for i, v := range words {
    fmt.Println(i, v)
}
```

range returns the next index and the value

i, v = 0..1..2.., words[i]



*This may be an **array**, **slice**, **string**, **map**, or a **channel***

summary

for

```
for i := 1; i <= 5; i++ {
  sum += i
}
```

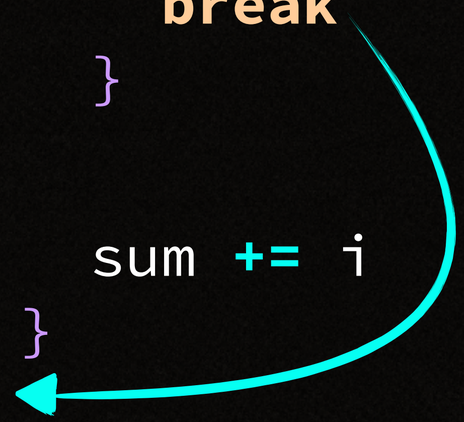
```
for ; i <= 5; i++ {
  sum += i
}
```

```
for i <= 5 {
  sum += i
}
```

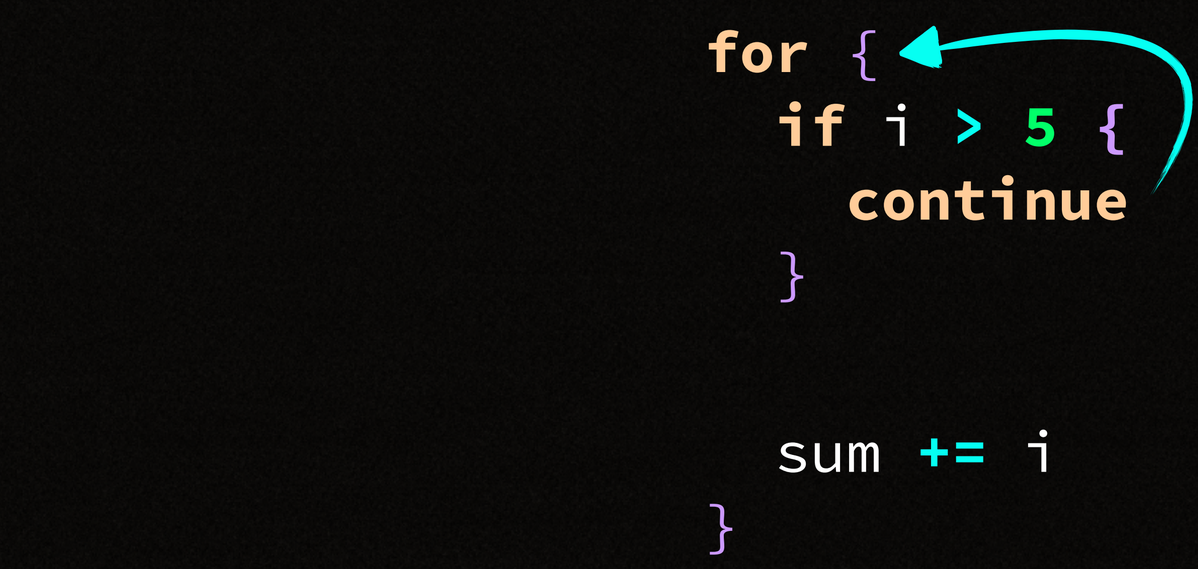
```
for {
  sum += i
}
```

break

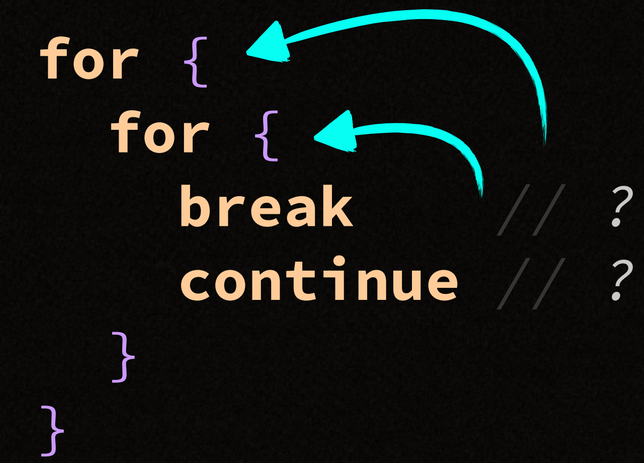
```
for {
  if i > 5 {
    break
  }

  sum += i
}
```



continue



nested loops

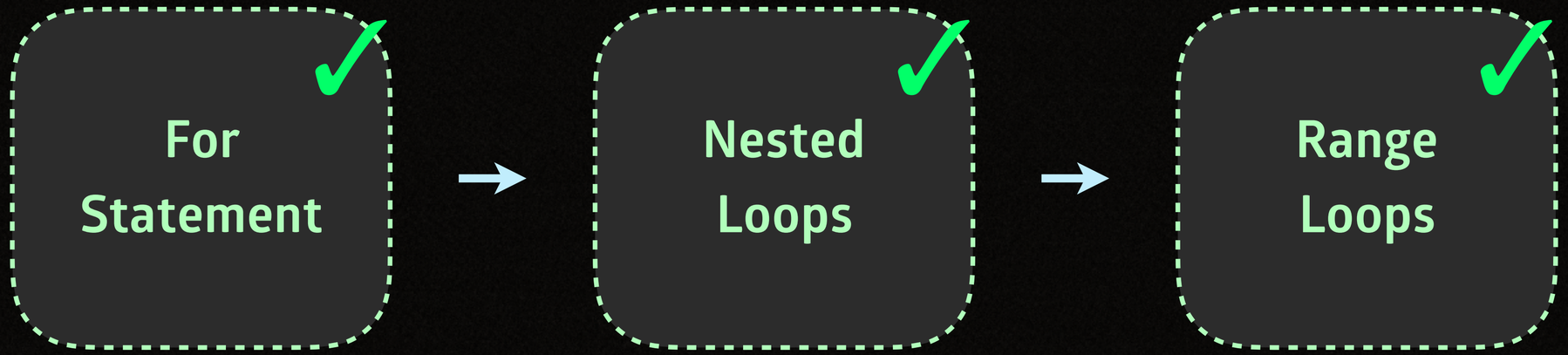


for range

```
for i, v := range os.Args[1:] {
}
```

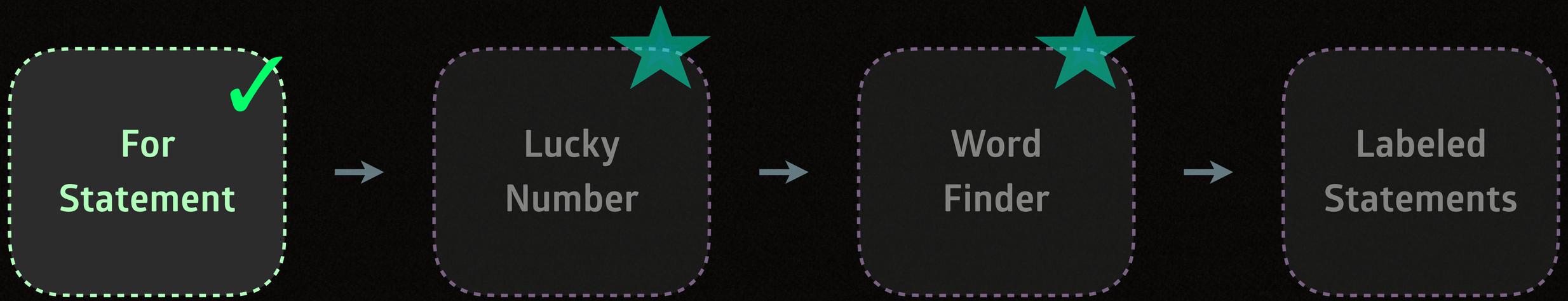
FOR STATEMENT

Congrats! You've learned everything about for statement!



LOOPS

First step is done!



🍀 LUCKY NUMBER

Project

Pick a number

Produce
random numbers

User wins when
`guess == random number`

RAND PACKAGE

rand = random

helps you to produce
random numbers

🍀 LUCKY NUMBER

Project

1

t = time.**Now**()

current time *as a* ***Time value***

2

t.**UnixNano**()

unix time *in* ***nanoseconds***

3

rand.**Seed**(*SeedNumber*)

seeds *the random number algorithm*

4

rand.**Intn**(n)

returns a ***random number*** *depending on the* ***seed number***

🍀 LUCKY NUMBER

Project

CHALLENGE!

#1 FIRST TURN WINNER

- Print a **special bonus message** if the player wins on the **first turn**

#2 RANDOM MESSAGES

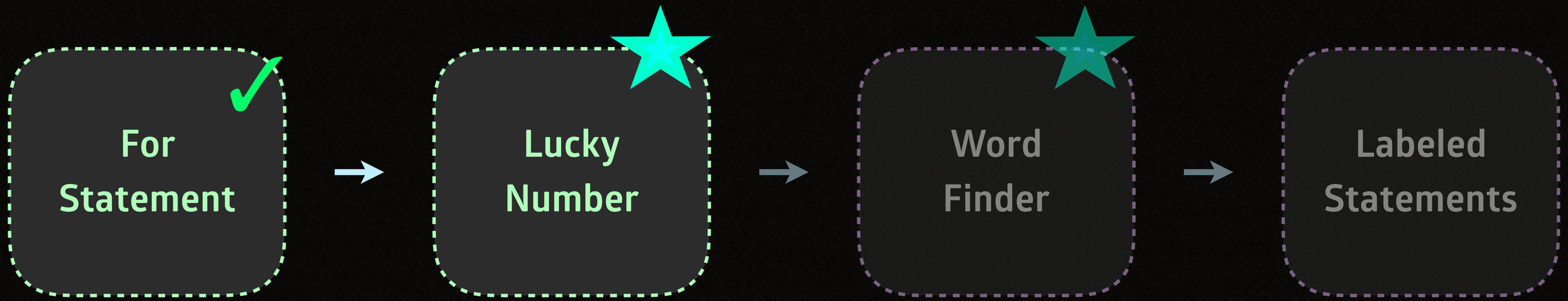
- Display **random won/lost** messages

#3 DOUBLE GUESSES

- Let the **player guess two numbers** instead of one
- When either of them picked by the computer print the winning message

LOOPS

Second step is done! Two steps to go!



🕵️ WORD FINDER

Simple Search Engine

words

{ "lazy", "cat", "jumps", "again", "and", "again", "and", "again" }

words

{ "lazy", "cat", "jumps", "again", "and", "again", "and", "again" }

query

{ "again", "and" }

words

{ "lazy", "cat", "jumps", "again", "and", "again", "and", "again" }

query

{ "again", "and" }

words

{ "lazy", "cat", "jumps", "again", "and", "again", "and", "again" }

skip the duplicates

query

{ "again", "and" }

CHALLENGE!

CASE INSENSITIVE SEARCH

Allow for **case-insensitive** searching

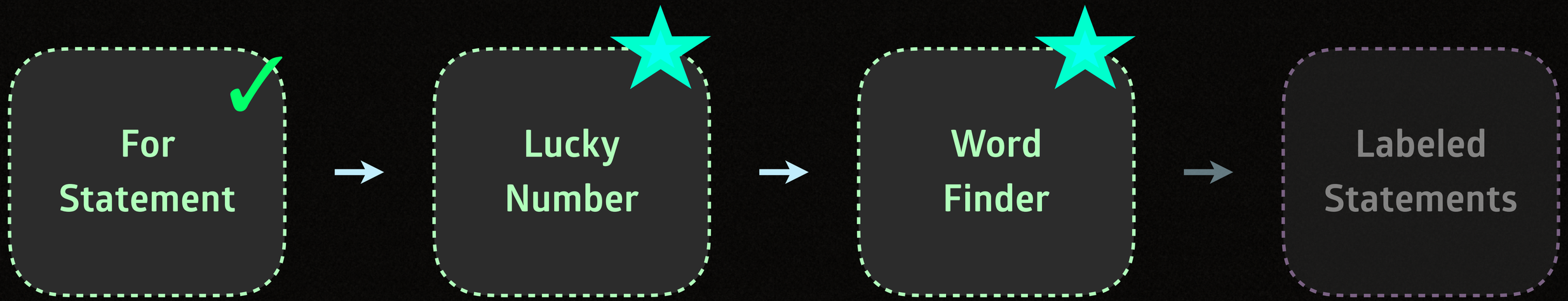
Let's say that the user runs the program like this:

```
go run main.go LAZY
go run main.go lazy
go run main.go lAzY
```

For all the cases above, the program should find the **lazy** keyword.

LOOPS

Almost done! Only one step left.



LABELED STATEMENTS

*Time to learn how to **control** the **execution** flow using the **labeled statements***

Labeled Break → Labeled Continue → Labeled Break & Switch → Goto

LABELED BREAK

Lets you terminate the statements that you've labeled

words

{ "lazy", "cat", "jumps", "again", "and", "again", "and", "again" }

query

{ "again", "and" }

words

{ "lazy", "cat", "jumps", "again", "and", "again", "and", "again" }

skips the duplicates

query

{ "again", "and" }

words

{ "lazy", "cat", "jumps", "again", "and", "again", "and", "again" }

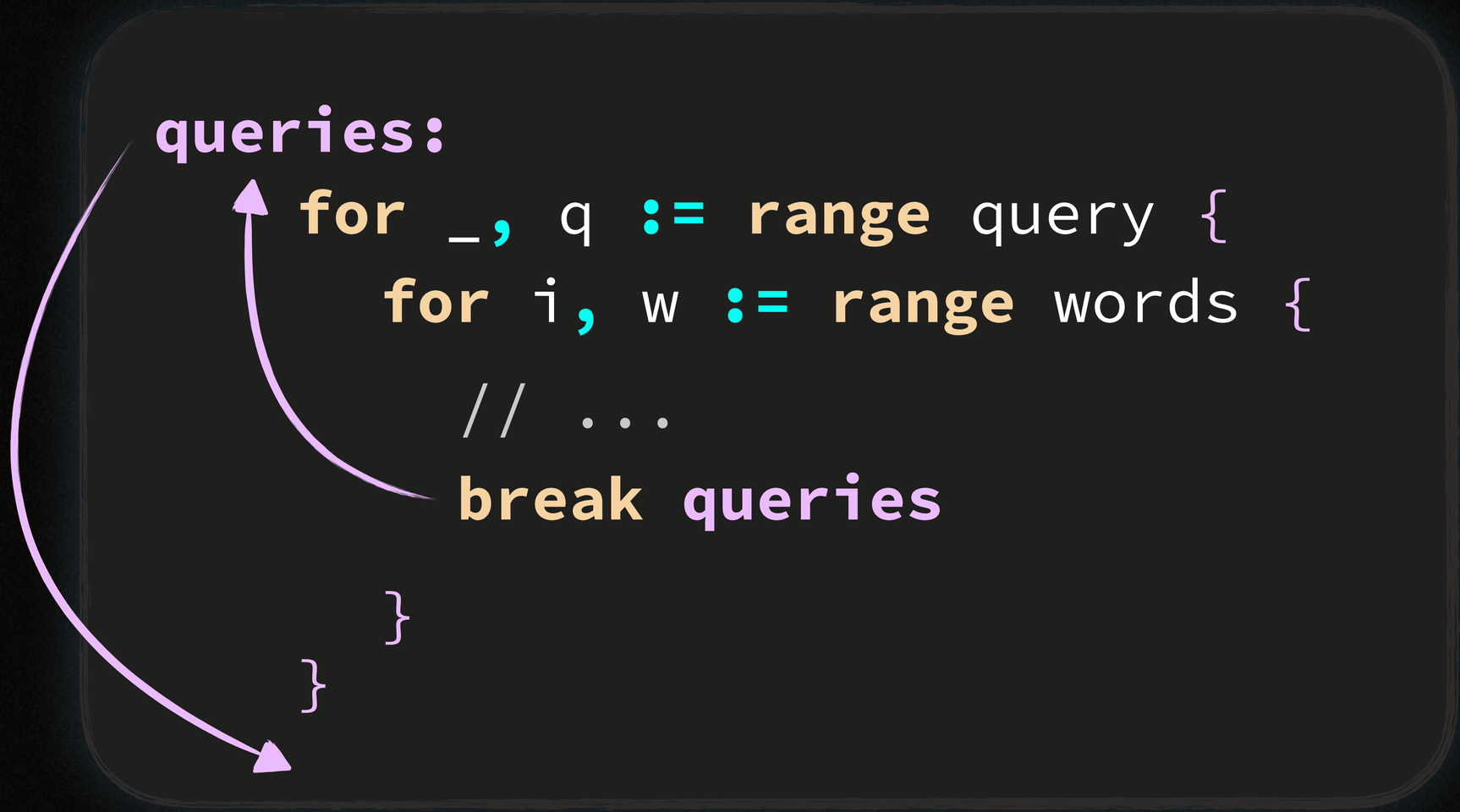
break the loop here

query

{ "again" "and" }

LABELED BREAK

Labeled break breaks the labeled statement



LABELED CONTINUE

Lets you continue the loop that you've labeled

words

{ "lazy", "cat", "jumps", "again", "and", "again", "and", "again" }

query

{ "again", "and" }

words

{ "lazy", "cat", "jumps", "again", "and", "again", "and", "again" }

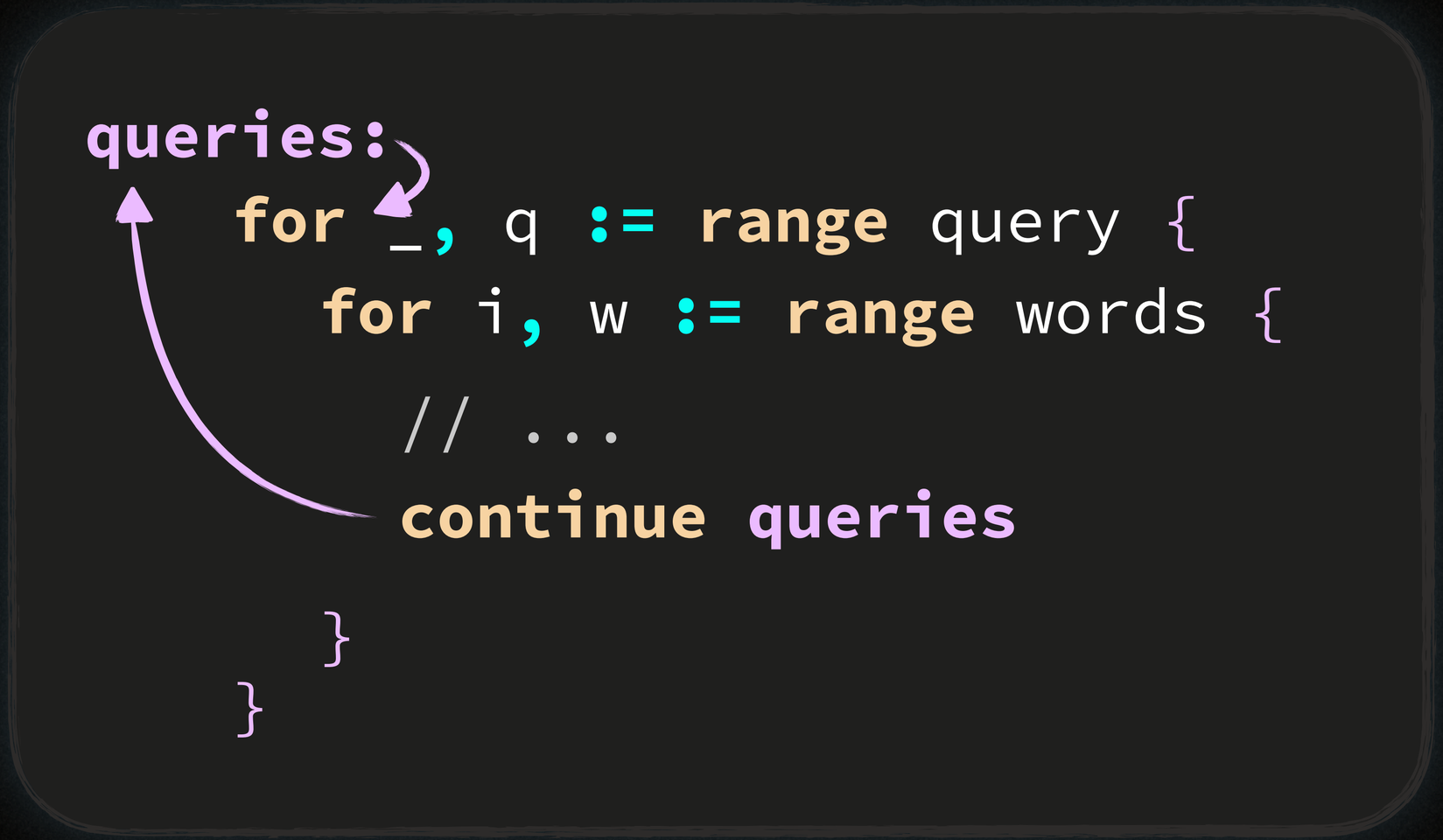
skip the duplicates

query

{ "again", "and" }

LABELED CONTINUE

Labeled continue** continues from the **labeled loop



LABELED BREAK IN SWITCH

It also works with a switch statement as well

query

```
{ "again", "and" }
```

words

```
{ "lazy", "cat", "jumps", "again", "and", "again", "and", "again" }
```

query

{ "again", "and" }

filter

"or", "and", "the"

words

{ "lazy", "cat", "jumps", "again", "and", "again", "and", "again" }

query

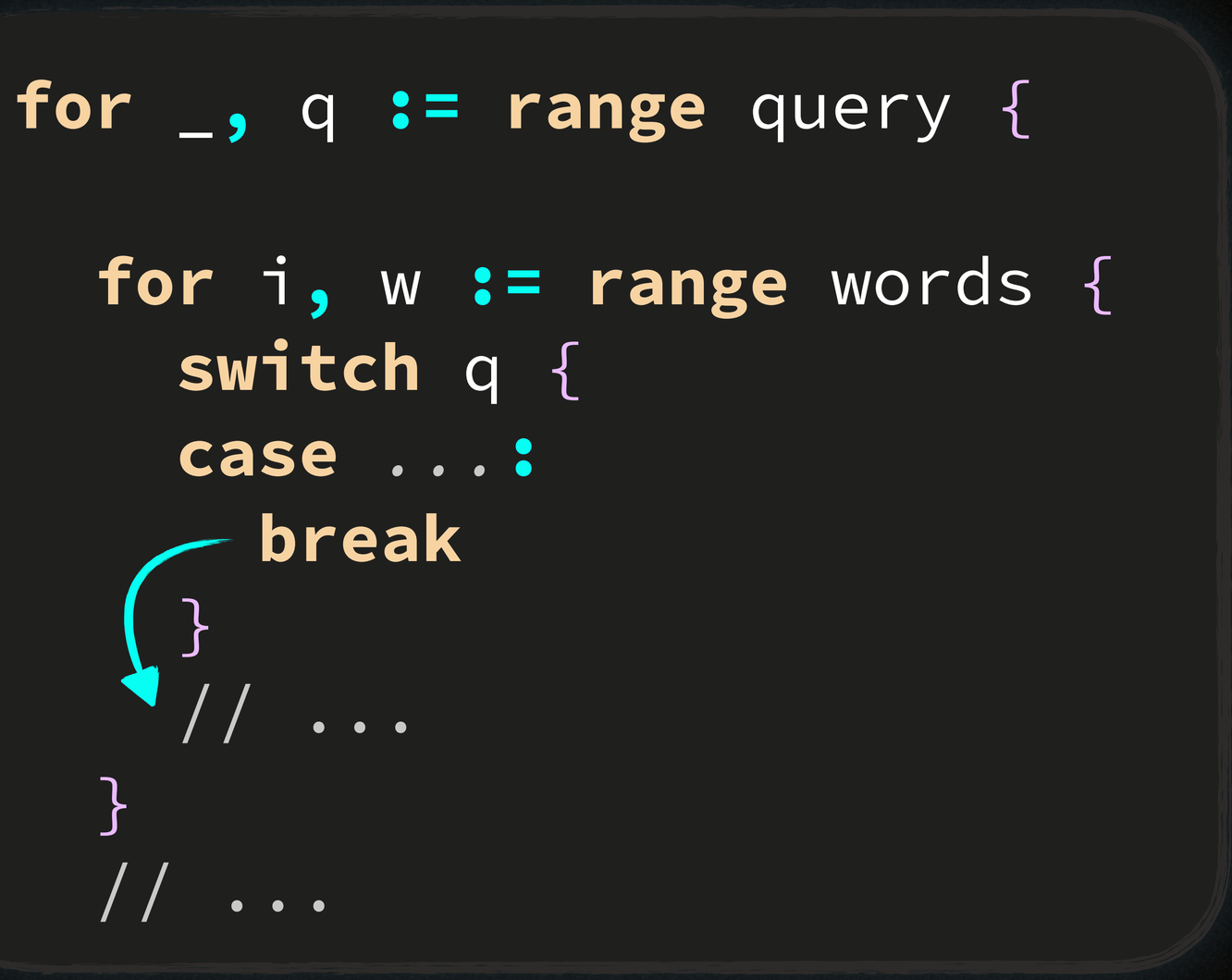
{ "again", "and" }

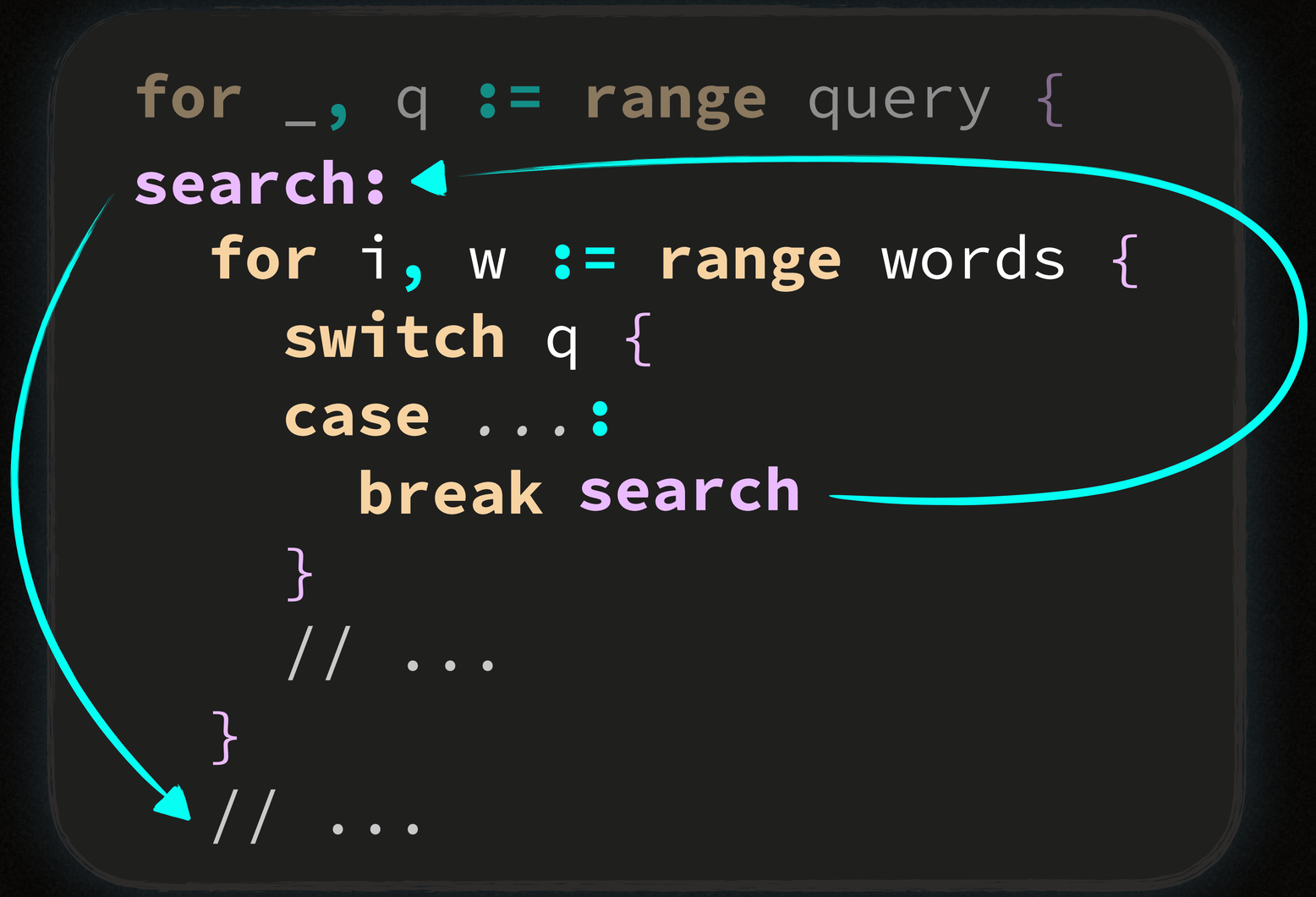
filter

"or", "and", "the"

words

{ "lazy", "cat", "jumps", "again", "and", "again", "and", "again" }


for _, q := range query {
for i, w := range words {
switch q {
case ...:
break
}
// ...
}
// ...


for _, q := range query {
search:
for i, w := range words {
switch q {
case ...:
break search
}
// ...
}
// ...

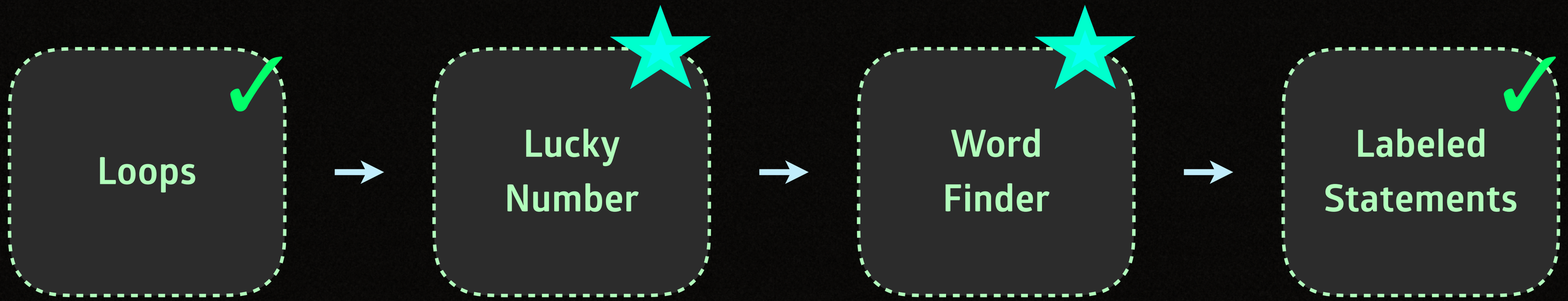
GOTO

Jump to a label

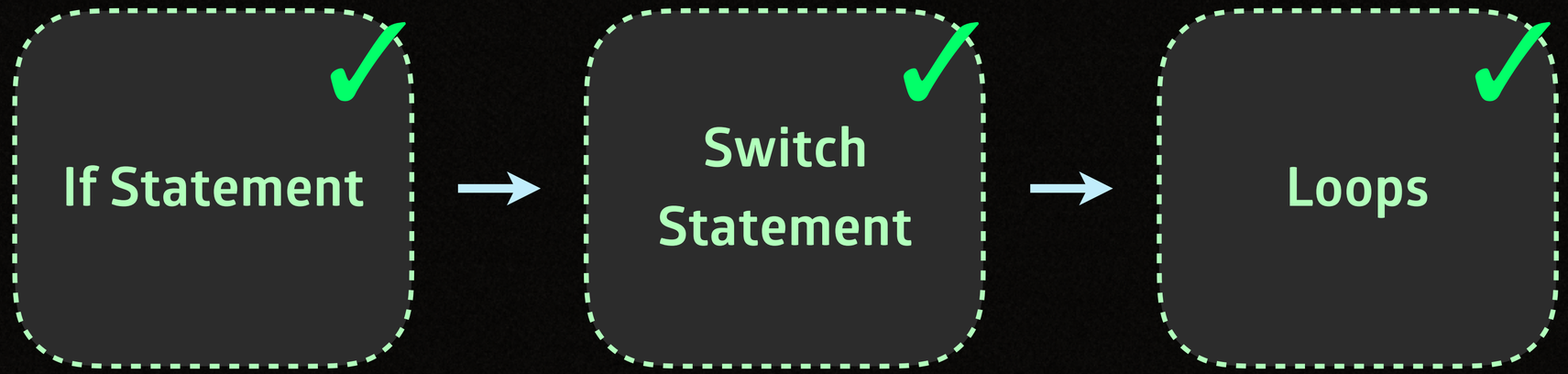
func someFunc() {
label:
// code
goto label
// code
}

LOOPS

Well done!



PART III

Fundamentals

Control Flow